CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2015; **27**:5037–5059
Published online 18 May 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3505

FireWorks: a dynamic workflow system designed for high-throughput applications

Anubhav Jain[1,*,†], Shyue Ping Ong[2], Wei Chen[1], Bharat Medasani[3], Xiaohui Qu[1], Michael Kocher[1], Miriam Brafman[3], Guido Petretto[4], Gian-Marco Rignanese[4], Geoffroy Hautier[4], Daniel Gunter[3] and Kristin A. Persson[1]

[1]*Environmental Energy and Technologies Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA*
[2]*Department of Nanoengineering, University of California San Diego, La Jolla, CA 92093, USA*
[3]*Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA*
[4]*Institute of Condensed Matter and Nanosciences (IMCN), European Theoretical Spectroscopy Facility (ETSF), Université Catholique de Louvain, Louvain-la-Neuve, Belgium*

SUMMARY

This paper introduces FireWorks, a workflow software for running high-throughput calculation workflows at supercomputing centers. FireWorks has been used to complete over 50 million CPU-hours worth of computational chemistry and materials science calculations at the National Energy Research Supercomputing Center. It has been designed to serve the demanding high-throughput computing needs of these applications, with extensive support for (i) concurrent execution through job packing, (ii) failure detection and correction, (iii) provenance and reporting for long-running projects, (iv) automated duplicate detection, and (v) dynamic workflows (i.e., modifying the workflow graph during runtime). We have found that these features are highly relevant to enabling modern data-driven and high-throughput science applications, and we discuss our implementation strategy that rests on Python and NoSQL databases (MongoDB). Finally, we present performance data and limitations of our approach along with planned future work. Copyright © 2015 John Wiley & Sons, Ltd.

Received 27 September 2014; Revised 3 March 2015; Accepted 14 March 2015

KEY WORDS: scientific workflows; high-throughput computing; fault-tolerant computing

1. INTRODUCTION

Scientists are embarking on a new paradigm for discovery employing massive computing to generate and analyze large data sets [1]. This approach is spurred by improvements in computational power, theoretical methods, and software, which have collectively allowed scientific computing applications to grow in size and complexity. Some applications have leveraged these new capabilities to increase system size, resolution, or accuracy via massive parallellism (e.g., in the simulation of combustion or climate). However, in other areas, researchers employ large amounts of computational power to increase the sheer *number* of calculations performed without significantly increasing the computational cost per calculation. We refer to this latter mode (requiring high concurrency of many smaller jobs), and particularly the case where jobs are continually added over long time frames [2], as "high-throughput" computing.

Such high-throughput calculations can require large amounts of total computing time; for example, the Materials Project (an effort to generate materials data through simulation) currently uses tens of

*Correspondence to: Anubhav Jain, Environmental Energy and Technologies Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA.
†E-mail: ajain@lbl.gov

millions of total CPU-hours per year, although each calculation is only approximately 100–1000 CPU-hours [3]. Practitioners of such high-throughput scientific projects face considerable challenges [4]. First, millions of simulations with complex dependencies must be executed and managed over long time periods and juggled among several computing resources. Second, failure tracking, recovery, and provenance are paramount to ensuring that all tasks are executed correctly. Third, these calculations must often run in a concurrent manner at supercomputing centers that are designed for executing small numbers of massively parallel simulations rather than very large numbers of smaller tasks. Finally, when the calculation procedure is nontrivial, parameter decisions must often be made algorithmically without explicit human intervention. Thus, full automation may require embedding expert knowledge within a *dynamic* workflow definition that automatically switches parameters and tasks depending on current and past results.

In this paper, we introduce FireWorks (FWS) [5, 6], an open-source workflow software tailored for high-throughput computation that has been used to perform over 50 million total CPU-hours of computational materials science calculations at the National Energy Research Supercomputing Center (NERSC). FWS is implemented in the Python language, which is widely recognized as one of the major tools in modern scientific computing, with extensive support for scientific computing in almost every domain [7–9]. As a high-level language, Python allows scientists (whose primary background is not in computing) to relatively easily extend FWS to suit the needs of their domain. Similarly, FWS is to our knowledge the first major workflow software to use a JavaScript Object Notation (JSON)-based approach to state management that utilizes MongoDB [10], an NoSQL datastore, as its database backend. We have observed that MongoDB—with its basis in JSON documents similar to Python's *dict* object—allows for direct mapping between workflow objects and their database representations in a very flexible way, while its simplicity enables nonspecialists to rapidly learn powerful features of the query language (e.g., regular expression matches) and feed the results into higher-level commands. In this paper, we describe how this approach allows FWS to support a unique combination of features found in other workflow systems [11–19] and introduce new ideas such as subworkflow duplicate matching based on JSON equivalence. Some of the capabilities of FWS include dynamic workflow programming, the ability to parallelize tasks within and across systems, and a comprehensive database allowing for advanced tracking, provenance, and failure recovery of jobs over long durations (e.g., years of continuous calculation). We note that many of these issues have been previously posed as important problems for future workflow systems [14].

2. OVERVIEW OF FIREWORKS

2.1. Overall infrastructure

At the most basic level, FWS can be separated into a datastore (*LaunchPad*) and clients (*FireWorkers*) that pull tasks (Figure 1). These two components are largely decoupled and interact through a *pull* model whereby FireWorkers request jobs from the LaunchPad. In many instances, this allows users to design workflows composed of universal tasks that are agnostic to the choice of client. In the simplest case, a FireWorker can execute an arbitrary workflow after the installation of FWS and any Python libraries that define custom user tasks. Strong coupling to produce more advanced behavior is possible (e.g., Section 3.1); however, the weaker coupling often enables a *write-once*, *execute-anywhere* paradigm that can be useful for scaling homogeneous computations across multiple clients.

Specifically, FWS employs the LaunchPad as a centralized MongoDB-based queue, from which jobs can be queried (based on a rich set of attributes) for execution at distributed resources. FWS implements this functionality at the client with a universal *rlaunch* script that (i) pulls a job from the LaunchPad, (ii) (optionally) creates a directory for the job, (iii) loads and runs the appropriate tasks, and (iv) updates the state of the job and its dependencies in the LaunchPad. Each execution of *rlaunch* completes a single step in the workflow.

The *rlaunch* script can run directly or wrapped in many ways, including daemon modes, modes to run on queues (Section 3.2), and modes to pack multiple jobs in parallel (Section 3.5). One could also envision using FWS as the workflow manager while using another system (e.g., a grid computing tool

DOI: 10.1002/cpe

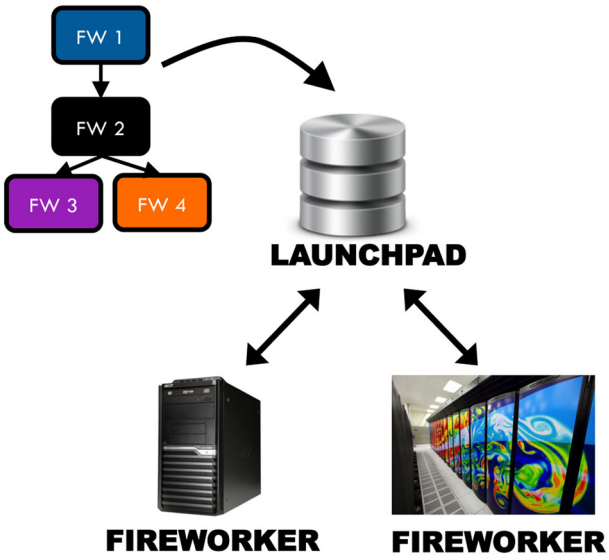

Figure 1. Basic infrastructure of FireWorks. Workflows composed of several jobs (Fireworks) are entered into a database (LaunchPad). Multiple computing resources (FireWorkers) can pull jobs from the LaunchPad and execute them.

such as Condor [16,18]) as the "execution manager" that strategically and repeatedly executes the *rlaunch* script over multiple resources in an opportunistic manner.

2.2. *Workflow model*

The FWS workflow model is illustrated in Figure 2. A *Workflow* in FWS is a directed acyclic graph (DAG) of individual *Fireworks* (jobs), where the edges represent control dependencies. As the graph is executed, side-effects of executing the Fireworks can modify the graph dynamically (Section 3.4). Each Firework consists of a sequence of one or more *FireTasks*, which are essentially single Python functions. For example, FireTasks bundled within FWS can call shell scripts, transfer files, write/delete files, or call other Python functions. A FireTask can return an *FWAction* (output) object

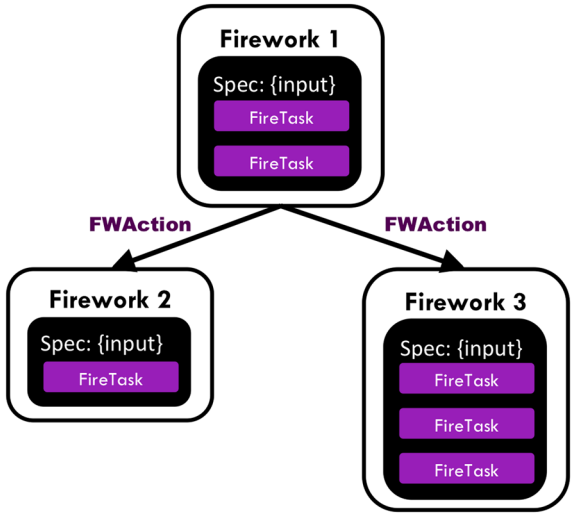

Figure 2. Representation of a workflow in FireWorks, which is composed of Fireworks consisting of multiple FireTasks. Each FireTask can return an output object (FWAction) that can pass data via JSON (either between FireTasks or between Fireworks) or modify the workflow dependencies.

DOI: 10.1002/cpe

that can store data, pass data, or dynamically modify its children depending on past output. The Firework also includes a *spec*, which is a JSON document that is passed into the tasks. For example, one can execute data-parallel projects by defining Workflows containing the same FireTasks but with different input data in the *spec*.

A Workflow can be represented by a JSON (or equivalently, YAML) document (Figure 3). One can compose Workflows via a Python interface (Code Example 1), by the command line (Code Example 2), or by directly loading a JSON or YAML file.

At the top level, a Workflow is composed of three dictionary entries: a list of all Firework definitions (*fws*), a mapping of their dependencies (*links*), and user-defined metadata (*metadata*) that can be used for querying and grouping Workflows.

Each Firework has an *id*, an optional *name*, and a dictionary *spec*. The spec contains two sections: the *_tasks* reserved keyword defines the list of FireTasks to be executed in sequence within the Firework. The remaining part of the spec is a general JSON document that is passed to the FireTasks to modify their behavior (e.g., to run the same functions over different data) and also allows for setting additional system-level keywords (e.g., *_priority*).

The Firework also stores information about its current state (e.g., COMPLETED and READY) as well as information about its execution on a FireWorker (*Launch*). The Launch object includes the following:

- hostname/IP address of the execution machine;
- the full state history of when the job was reserved in the queue, started running, and completed; and
- the FWAction returned, including stored data defined by the user and any dynamic operations.

Other than basic dependencies, the JSON workflow specification contains no control logic (if–then, for-loops, etc.). Control (iteration, branching, and adding additional subworkflows) is achieved by returning an appropriate FWAction (discussed in Section 3.4). Because Python code can be used to construct the FWAction, the control logic can be arbitrarily complex.

The LaunchPad is implemented using MongoDB [10], an NoSQL database that is ideal for storing and querying binary JSON (a variant of JSON) documents. Because all objects in FWS can be encoded as JSON, no object-relational mapper is needed to translate FWS objects into MongoDB. Full-featured search of any item in the JSON representation of the Workflow is possible through MongoDB's built-in query language. For example, one can query using regular expressions, parameter ranges, values in a list, entire subdocuments, or even write a query in JavaScript. Furthermore, these queries can be fed into higher-level commands to rerun or cancel Fireworks. For example, one could rerun all Fireworks with a particular parameter range in the spec or cancel all jobs with a particular type of FireTask. Thus, rather than set up a single "set up and execute" pattern, one can maintain and update workflows fluidly over the course of years.

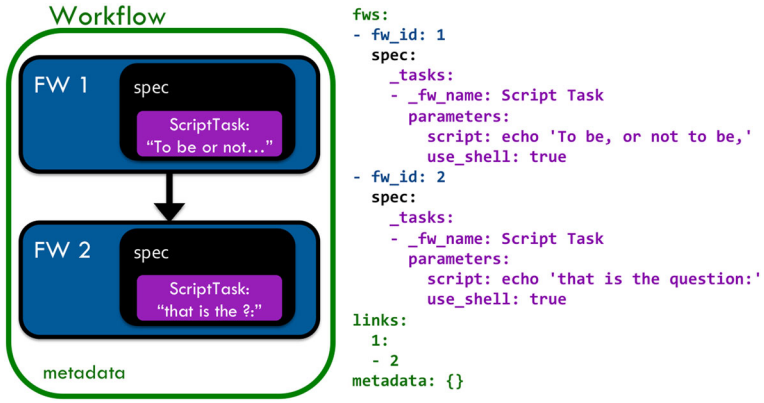

Figure 3. Representation of workflow (left) and corresponding YAML representation (right). For clarity, aspects such as job state, launch data, metadata, and optional names are not presented.

DOI: 10.1002/cpe

3. DESIGN AND IMPLEMENTATION

Next, we summarize some of the major features of FWS. We expect that the number of features will grow with further development of the FWS code, and a full list can be found in the official documentation [6].

3.1. Worker-specific settings

A basic feature required by many applications is the ability to selectively map jobs to clients, as well as to account for small differences in the runtime environments of different computing resources. In FWS, each FireWorker can specify a configuration with one or more *category* labels, which will restrict the Fireworks being pulled to those with the matching category in the Firework spec. More generally, the FireWorker can specify a custom MongoDB query for selecting Fireworks to pull; in principle, this allows for complete control to match jobs to resources. However, we note that this matching is explicit and less flexible than more complex systems (e.g., as in Condor's "matchmaking" service [18]).

FireWorkers can also specify a set of "environment variables" (termed *FWEnv*) in their configuration; these variables can be set to JSON objects. FireTasks can be programmed to refer to these variables (or reasonable defaults) in their execution. For example, the FWEnv might specify the path to a particular script on different systems, and the FireTask would refer to this value when executing the code. Thus, one can often maintain the abstraction of "write-once, run-everywhere" even when system-level differences must be taken into account.

3.2. Running on queueing systems

When executing jobs on a large computing cluster, job requests must often pass through a queueing system. In the most basic use case, this is very straightforward with FWS. One needs only to submit a standard queue script in which the executable is the *rlaunch* command that pulls and runs a job. The *rlaunch* command itself contains options for running a single job (*singleshot*), consecutively running many jobs (*rapidfire*), or packing jobs consecutively in time and across cores (*mlaunch*—Section 3.5). No special setup is required besides having the FWS installed and having a network connection to the LaunchPad (Section 3.10). Because the queuing system is unaware of the presence of FWS and vice-versa, no action needs to be taken if there are problems with the queue.

In addition to this basic operation, FWS contains several routines specifically intended to facilitate the process of running on queues. For example, FWS provides a QueueLauncher that can submit jobs automatically and maintain a set number of jobs in the queue over long time periods. The QueueLauncher operates based on a QueueAdapter, which contains a template for the queue script, a command to submit jobs, and a function to get the number of jobs in the queue. At this time, QueueAdapters are implemented for PBS/Torque, Sun Grid Engine, SLURM, and IBM LoadLeveler. Adding new QueueAdapters is simple, and the framework is general enough to easily incorporate more abstract systems such as the NERSC Web Toolkit (NEWT) [20] Representational State Transfer (REST)-based [21] submission at NERSC.

FireWorks also lets jobs specify details of their queue submission. This is useful when one has heterogeneous jobs requiring different walltimes, number of processors, and so on. This system is called "reserved queue launching" and operates as follows. First, the QueueLauncher reserves a Firework to run in advance of queue submission. Next, it uses information inside the *_queueadaptor* reserved keyword in the Firework spec (set by the user) to override parameters in the templated queue submission script and replaces the generic *rlaunch* command with an instruction to specifically pull and run the reserved Firework. Finally, the QueueLauncher submits this modified queue script that is tailored to a specific job. Thus, in effect a user can define heterogeneous queue settings for different Fireworks at the time of creation. Reserved queue launching tracks and stores additional properties of a Launch such as the queue id and time spent waiting in the queue.

DOI: 10.1002/cpe

3.3. Fault tolerance and failure detection and recovery

Fault tolerance is essential when running millions of workflows over multiple clients over long time periods. FWS is built on the premise that jobs will fail for myriad reasons and accordingly provides several routines for detecting and correcting such failures. The types of failures addressed by FWS are "soft" failures (script throws an error), "hard" failures (machine unexpectedly fails), and queue failures.

The easiest type of failure to address is a "soft" failure whereby the FireTask internally raises an error (or a script returns a nonzero error code). FWS catches such errors and uses them to mark the job state as FIZZLED (failed). Because many details of the output (including the stack trace) are stored in the LaunchPad, the user can search for and further examine these jobs (including the error message and launch directory) via built-in monitoring tools, make the appropriate changes, and rerun the affected Fireworks and their dependencies.

Hardware failures or memory overload can kill a job without warning ("hard" failure). To detect these failures, FWS uses a separate thread to send a heartbeat, that is, to *ping* back the LaunchPad with an "alive" status (Figure 4). A job that has failed catastrophically will also stop its heartbeat. FWS can detect such "lost runs" and provides commands to mark such jobs as FIZZLED or rerun them. Thus, even catastrophic events can be identified and be subject to targeted reruns.

Finally, when working with queueing systems, it is possible that the job can be "lost" in the queue. For example, the administrators of a supercomputing center might flush the queue before maintenance. This situation only affects jobs submitted in "reservation mode" (Section 3.2), for which running through the queue is made explicit. FWS provides a routine whereby users can resubmit jobs that appear to be stuck in a queue for longer than a set period (e.g., several weeks).

3.4. Dynamic workflows

As automation is applied to more complex problems, a major scientific use case is to embed intelligence within the workflow definition. One of the more unique aspects of FWS is its implementation of dynamic workflows, which allow FireTasks to modify the Workflow based on their execution in a way that cannot be predicted during the initial creation of the Workflow. A simple example would be a looping Workflow or iterative solver in which additional Fireworks are created to repeat a calculation (while increasing some parameter in the JSON spec) until reaching a nonobvious stopping point. A more complex example would be a Workflow that evaluates the result of an initial test calculation to automatically develop the remainder of the workflow based on the result (perhaps following one of several predefined paths). Thus, with dynamic workflows, the user need not define the full Workflow at the time of creation, instead letting the jobs amend the

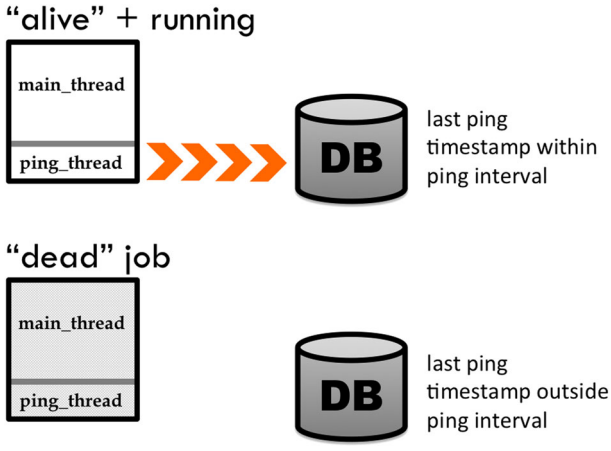

Figure 4. Method for tracking down difficult-to-detect failures: a ping thread periodically provides positive confirmation of a job's "alive" status via a heartbeat.

DOI: 10.1002/cpe

definition during execution. We note that similar ideas based on the formalism of Petri nets have previously been developed for dynamic error-handling [22].

Dynamic workflows are defined through FWActions, which are in turn returned by FireTasks. The FWAction is interpreted upon completion of the launch. The possible FWActions (which can be specified alone or in combination) are presented in Figure 5 and include the following:

- *update_spec*, which allows the current Firework to push changes to the spec of the children Fireworks via JSON. This is most commonly used for information passing; at the most basic level, one can think of the root keys in the returned document as "ports" for passing data, with the values being the information to be passed as JSON.
- *mod_spec*, which is a more complex version of update_spec that uses a MongoDB-like update language to allow for complex JSON modifications such as pushing to an array.
- *defuse_children*, which cancels child jobs. The defused jobs can later be restarted if desired.
- *exit*, which serves as a break-style statement that skips the remaining FireTasks in the current Firework and continues execution of the Workflow.
- *stored_data*, which accepts a JSON document to store in the LaunchPad for future querying and display.
- *additions*, which can create new Workflows to execute. When used in conjunction with *defuse_children*, it can act as a branching operation.
- *detours*, which performs additional Workflows before executing the original children. For example, a detour can be used to handle an exception and implement a "self-healing" workflow.

An example of a dynamic Workflow that counts Fibonacci numbers until reaching an arbitrary (and unknown) stopping point is presented in Figure 6. Initially, the Workflow contains only a single Firework; however, this Firework will create its own child, and the process will repeat *ad infinitum* until the stopping criterion is met (Code Example 3). At the end of the run, the Workflow will contain many more Fireworks than its starting point.

3.5. *Job packing*

Many high-performance computing centers are designed to run and manage small numbers of massively parallel jobs. This is reflected by queued job limits (usually less than a dozen) and minimum job sizes that prevent running in a high-throughput manner. Unfortunately, such limits prohibit running high-throughput computing in which millions of individual jobs must eventually be executed.

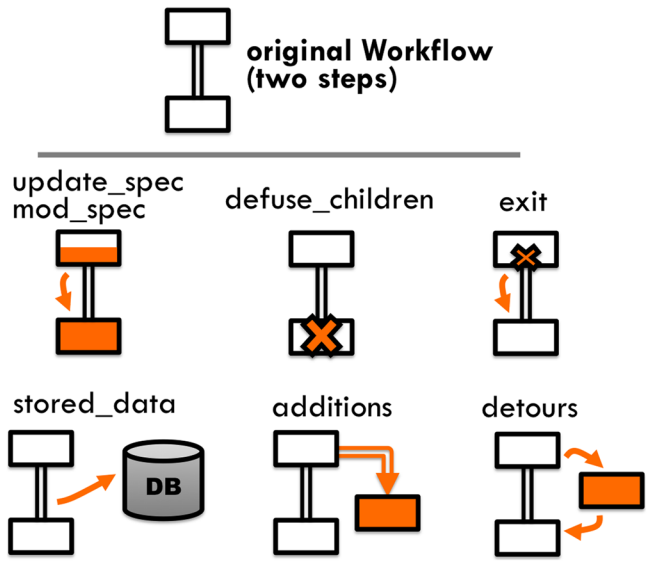

Figure 5. Types of dynamic FWAction outputs supported by FireWorks. The orange sections correspond to dynamic actions; see text for additional details.

DOI: 10.1002/cpe

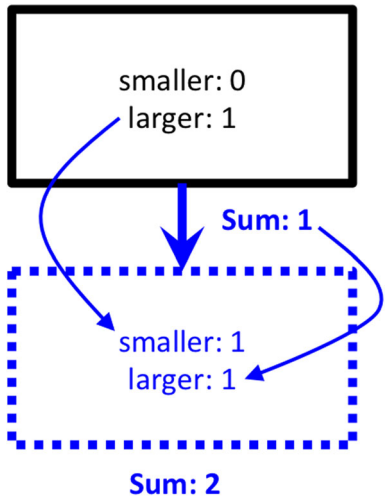

Figure 6. Representation of a dynamic workflow to count Fibonacci numbers; see the text and Code Example 3 for additional details.

FireWorks includes a built-in ability to pack many small jobs into a larger ensemble that can effectively use a large number of computation nodes/cores. No effort is required to specially design workflows for this purpose; the packing occurs automatically through modifications of the script used to pull and run jobs. Job packing allows the user to explore parallel opportunities at the organization level in a way that can supplement job level parallelism (e.g., OpenMP/MPI) seamlessly. The "packing" can be multiple single-core jobs into a single multicore processor, multiple parallel jobs over multiple nodes, and multiple serial jobs over multiple nodes. The last case is handled simply by calling the launching script via MPI (this will set up multiple processes that each pull a job and run it). The former two cases are handled using Python's multiprocessing module and are described next.

Figure 7 depicts the execution flow of job packing. Job packing is simply a wrapper of the "non-packed" *rlaunch* script to pull a job and execute it. However, rather than performing this action once, it uses the multiprocessing module to perform this action in parallel. For example, one can have each core of a multicore processor pull a job and run it, thus automatically running multiple serial jobs in parallel on a multicore machine. We emphasize that no special setup is needed when defining the jobs; rather, one must only change the execution script on the FireWorker.

We note that this mode can also run jobs with a low level of parallelism (e.g., 16 cores) on an allocation with multiple nodes (e.g., 4 nodes with 16 cores each). In this case, one would start multiprocessing with N processes, where N is the number of *nodes*; each process will pull a Firework and run it. During execution, when the "mpirun" (or equivalent) command is invoked, the job will be passed to an available node automatically via MPI. Thus, up until the point when "mpirun" is invoked, multiple processes are running on a single node to query a job and begin execution. After invocation of "mpirun", each job is automatically run on a different node.

We note that to limit the number of database requests sent to the database server simultaneously, we have introduced a DataServer object that behaves as a proxy/buffer for job queries. The DataServer is the only process that can issue job query requests to the database server; all other processes forward requests to the DataServer via a socket. This helps the multiprocessing to scale to reasonably large numbers without overloading the systems (e.g., approximately 100 processes are currently supported without issue).

3.6. Automated duplicate checking at the subworkflow level

When executing hundreds of thousands or millions of workflows, a natural use case is to avoid running the same tasks twice across workflows and within them. For example, if several users independently submit workflows, one would seek to detect the duplicated steps of these workflows and only run

DOI: 10.1002/cpe

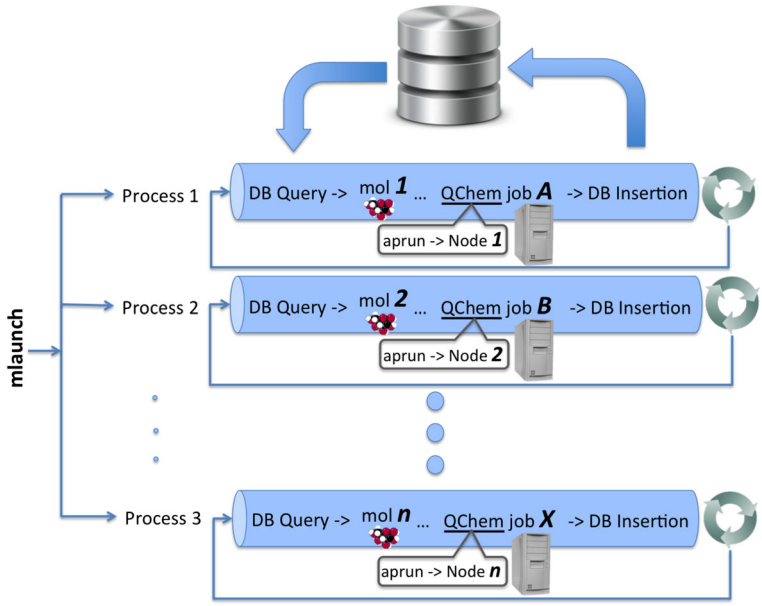

Figure 7. Schematic of job packing in the context of computational chemistry. The "mlaunch" command creates multiple processes; each process queries for a job to run (mediated by the DataServer, not shown) and executes it. If the job includes a parallel command such as "mpirun" or "aprun", each MPI task can be run on a separate node. In this diagram, the code is automatically parallelizing over different molecules to run the Qchem [46] electronic structure code.

them once. With FWS, such *automated* duplicate detection is available at the *sub*workflow level and falls naturally out of the framework. Thus, FWS can power a distributed execution system whereby multiple users submit their individual workflows and redundancies are automatically eliminated.

Because Workflows in FWS can be represented as JSON documents, it is easy to check for equivalence. Two Workflows mapping to the same JSON document will produce the same result, provided that the jobs are themselves deterministic (the same input reliably produces the same output). This idea holds at the individual Firework level as well: two Fireworks that map to the same JSON document will execute the same codes (_tasks) using the same input JSON (spec), and thus produce the same output. This feature, in conjunction with the fact that the full FWAction output is also stored as a JSON object, allows FWS to naturally support duplicate checking at the subworkflow level.

The basic operation of duplicate checking is as follows: if enabled, a Firework (before running) searches for another Firework in the database with the same spec (a generalization of this statement follows later). If a matching Firework has already completed or is running, the job is not launched. Rather, the launch data of the duplicated job are copied and applied to the current Firework. The launch data encapsulate both the runtime information (directory, machine, etc.) and the full output of the matching Firework. Note that this procedure not only skips execution of duplicated Fireworks but also applies dynamic actions, stored data, and spec updates as needed for passing to child Fireworks. Thus, we have completely simulated running of the duplicated Firework without applying any additional computing.

In FWS, Workflows are automatically checkpointed at the Firework level (they can be rerun, canceled, reprioritized, etc. at that level). We note that duplicate checking also operates at the Firework level and often allows for even more powerful schemes than checkpointing. For example, a new Workflow could alternate between novel and previously run Fireworks, and duplicate checking would sort out what requires computing resources and what data can be copied from previous runs.

Finally, we note that in many instances, *exact* matching of the JSON documents is too strict of a definition for duplicate detection. For example, a user might want to avoid duplication within certain numerical tolerances of the input or without regard to optional/metadata parameters in the spec.

DOI: 10.1002/cpe

FWS allows the user to write custom *DupeFinder* objects in Python that allows full flexibility in defining "equivalence" between documents.

3.7. Automated job provenance

When running many jobs over a long period, one point of concern is reproducibility and provenance; that is, how well can we trace back the execution of a past run or track down a bug? By its design, FWS automatically stores many of necessary ingredients for successful provenance. For example, the LaunchPad automatically stores the codes to be executed (the FireTasks), what parameters those FireTasks were initialized with, and the full input dictionary used by the FireTask (the spec). FWS also stores information about job execution, including the machine hostname and IP address, start time, end time, and stack traces of errors. All this information is available as structured JSON that can be queried via MongoDB and without any additional effort from the user.

Note that even when jobs are rerun, an archive of the previous launches is maintained by default. Thus, one can readily see information on what occurred during past runs (including stack traces of errors). There is also an option to archive a Workflow, which acts as a "soft" delete that maintains all the provenance information (e.g., FireTasks and input spec) but prevents further runs or duplicate checking.

Note that while this data usually allows for "practical" tracing back of job execution, it is by no means complete. For example, versions of code are not automatically stored (nor are the codes themselves). However, we note that metadata such as code versions can be added manually to both Fireworks and Workflows such that more detailed provenance is possible with user input.

3.8. Frontend interface and job reporting

While most common tasks in terms of adding, monitoring, and maintaining Workflows can be performed using command line tools, FWS also includes a basic frontend interface built on Flask [23]. Because Flask includes a web server implementation, one can launch the frontend without installing additional software. Currently, the frontend only reports data on Workflows and cannot be used for adding jobs or modifications (e.g., reruns, failure detection, and priority modification).

The main page of the frontend (Figure 8) provides a summary report of the FWS database and the status of all Workflows. In addition, the detailed state of all Fireworks within the newest Workflows is listed in a color-coded way. In addition to the summary report, users can click various options (e.g., the fw_ids or the state) to obtain more information and browse, for example, all FIZZLED Fireworks. The advantage of the frontend is its ease of use and the possibility to allow multiple users to see the status of Workflows via the web without having to install FWS or become familiar with the command line tools.

FireWorks also provides a convenient reporting package to acquire run statistics. Instead of processing large datasets locally, the reporting package builds on the highly optimized MongoDB aggregation framework, which aggregates data through a streamlined multistaged pipeline on the server. As a result, it is very efficient to generate summaries of workflow completion statistics for a specified time range. One can also readily compile a progress report for job execution on a daily basis.

The reporting package is also a valuable tool to troubleshoot and optimize workflows and computing resource problems. For example, one can debug failed jobs by aggregating over a property in the spec to detect if a particular type of job (as defined by its spec) fails more often than others. One can also "identify catastrophes" by reporting days with high failure rates of jobs, perhaps related to hardware failures on the computing resources. These functionalities are particularly useful when attempting to make sense of execution over millions of heterogeneous jobs and several workers. The reporting features are currently available via the command line, and we plan to incorporate these features into the frontend interface in the future as rich graphical data.

3.9. Tracking output files

An important consideration when running concurrent jobs is the ability to quickly check the status of raw output files both during and after execution. For example, one might want to check the progress of several jobs in real-time or examine the last few lines of output in all jobs that have recently failed.

DOI: 10.1002/cpe

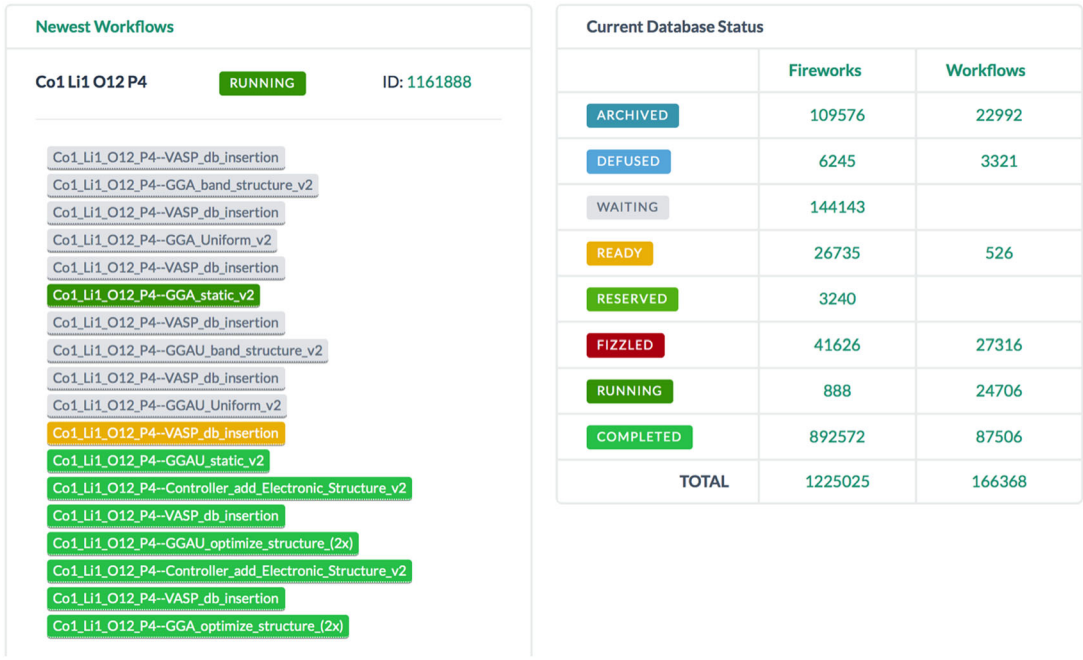

Figure 8. Screenshot of FireWorks frontend.

However, with potentially thousands of jobs running simultaneously and output files scattered across multiple resources, it becomes difficult to achieve this in an effective way.

FireWorks provides a file tracking feature whereby the first or last few lines of one or more files can be relayed periodically to the LaunchPad. Command line tools allow one to lookup these outputs from a central location. Thus, one does not need to log in to the resources where job execution is occurring in order to track outputs. Job tracking occurs as a lightweight execution within the ping thread process described in Section 3.3. By pinging more frequently, one can get more frequent updates regarding job output files (however, with a penalty of increasing database traffic).

3.10. Offline access

One limitation of FWS is that FireWorkers need to be able to connect to the central database to pull a Firework to run. In most cases, we expect such a network connection to be available; however, a client might not have access to a network connection or be firewalled in some instances. This can indeed be the case in supercomputing centers, where "compute" node communications can be restricted to an internal network.

One option in such cases is to host the LaunchPad within an internal network free of firewalls. However, FWS also provides an "offline" mode whereby jobs are *pulled* by a "master" entity with a network connection (e.g., a login node) and subsequently serialized to local files (Figure 9). The compute nodes can instantiate the job from these files and write all their output communications (pings, FWAction outputs) to a different file. The master then periodically checks the serialized outputs to update the LaunchPad's version of the data (including dynamic actions), thereby closing the loop.

4. PERFORMANCE

A major consideration when evaluating workflow software is performance overhead associated with the software. In FWS, sources of overhead include (i) time to query a job from the database before running, (ii) time to update the central database and dependent workflow statements after running, and (iii) time for internal processes such as loading the FireTasks. These times will in turn depend on factors such as network speed and size of each job document (e.g., a very large spec involves more data transfer).

DOI: 10.1002/cpe

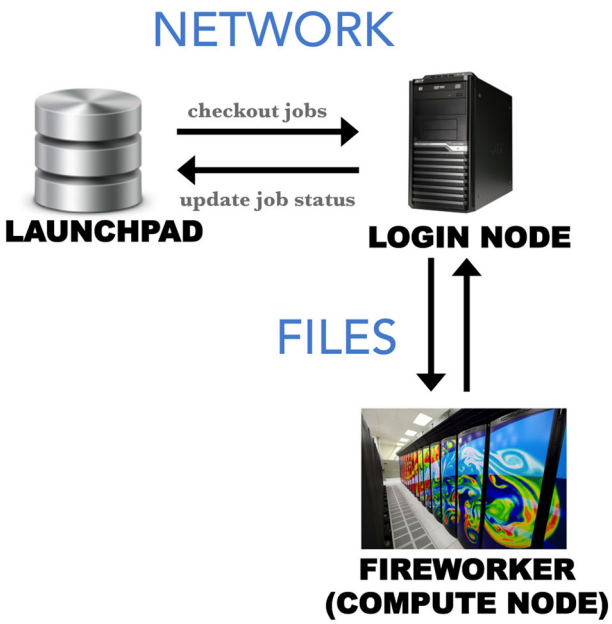

Figure 9. In the case where compute nodes cannot access the LaunchPad because of network restrictions, FireWorks provides an offline mode whereby communication is handled by a mixture of files and communication with a login node that has network access.

We ran experiments on the NERSC "Carver" system (two quad-core Intel Xeon X5550 "Nehalem" 2.67 GHz processors giving eight cores/node, and 24 GB of memory) with the LaunchPad hosted on the internal NERSC network using FWS v0.96. We were interested in testing the scalability of large workflows of varying types. Thus, we tested four types of Workflows: reduce, sequence, parallel, and pairwise-dependent (as illustrated in Figure 10) that employed between 200–1000 Fireworks per Workflow. Each Firework was a "no-op" that exited immediately, such that the measurements are essentially the overhead of FWS itself.

The results are presented in Figure 11 and plot the average time per task as a function of workflow size. The results are nearly or completely linear for all workflow types when there are five workflows, and in all cases, overhead are fractions of a second per Firework. This scaling behavior is achieved by just-in-time loading of Fireworks within the Workflow so that unnecessary data are not fetched during Workflow updates, as well as by caching some Firework state information in the Workflow itself. We note that prior to introducing these enhancements, preliminary performance tests had revealed an O (N^2) (where N is the number of Fireworks in a Workflow) scalability issue for large Workflows.

Figure 11 indicates that when there is only a single workflow in the database, the time per Firework increases roughly linearly with the number of tasks because of increased time spent loading the

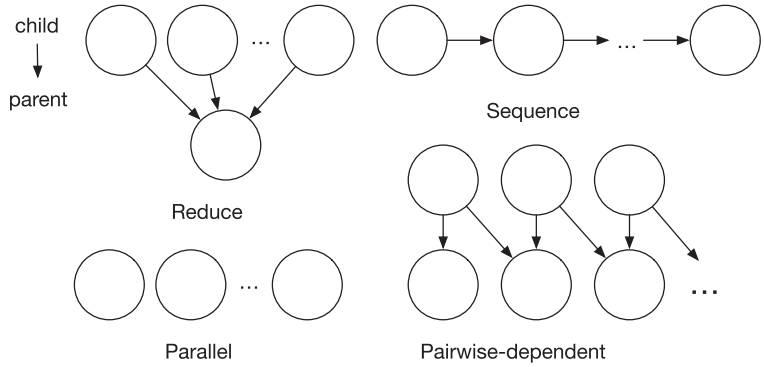

Figure 10. Workflow patterns used to test performance: reduce, sequence, parallel, and pairwise dependent.

DOI: 10.1002/cpe

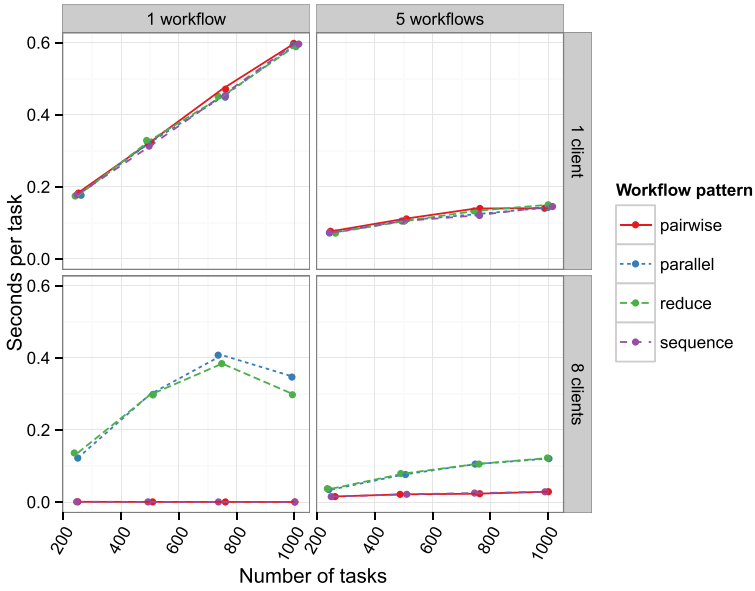

Figure 11. Results of performance testing; the overhead per task/Firework is always significantly less than 1 s, even for Workflows consisting of 1000 Fireworks. In more typical high-throughput situations (multiple workflows, multiple clients/FireWorkers), the overhead can be less than 1/10 of a second per job.

workflow during updates. As expected, adding multiple clients (FireWorkers) for a single workflow is helpful only when the workflow itself contains inherent parallelism, as in the "reduce" and "sequence" execution patterns.

When there exists multiple workflows in the database and a single client, the overhead per task is considerably reduced to less than 0.2 s even for larger workflows. This is because FWS uses a workflow-level lock to prevent race conditions during updates. When multiple workflows are present, the client spends less time waiting for a common lock. Introducing more clients only improves the situation; however, it is unclear at this time why the "pairwise-dependent" and "sequence" workflows (with less inherent parallelism) outperform the "reduce" and "parallel" workflows.

4.1. Job packing and overall throughput limits

To test the throughput limits of FWS with a basic configuration of a single database node, with and without the "job packing" mode (Section 3.5), we ran sets of between 1 and 1024 concurrent workflows within a single queue submission and measured the total time spent. Each workflow was again set as a "no-op" with zero execution time. The tests were again run on a single "smallmem" worker node on the NERSC Carver system and pulled workflows from a remote database node (six dual-core Intel(R) Xeon 2.70 GHz processors, giving 12 cores/node, and 64 GB of memory) in Berkeley separated by a network RTT of ~3 ms. For each set of workflows, we reset the database, loaded the requisite number of workflows, and timed the execution in both nonpacked ("rlaunch") rapidfire mode and the job packing ("mlaunch") mode. The job packing launches N workflows in parallel (Section 3.5); in all cases, we set N to the total number of workflows for full concurrency.

The results (shown in Figure 12) demonstrate that the throughput saturates between 5 and 6.5 workflows/s (300–400/min), with job packing improving performance by roughly 20%. While this performance is certainly limited, it still represents a vast improvement over the time taken to launch workflows through a batch queue. Scaling up the database may provide a path to higher throughput.

As a general rule of thumb guided by the ~6 jobs/s result, we expect that the utilization will be high when the *level of concurrency is kept at or below the number of seconds per task*. For example, for 100-s tasks, utilization should be high for 100 or fewer concurrent tasks. The use cases that have employed FWS thus far (discussed in Section 5) fall within this range, with typical job times in the

DOI: 10.1002/cpe

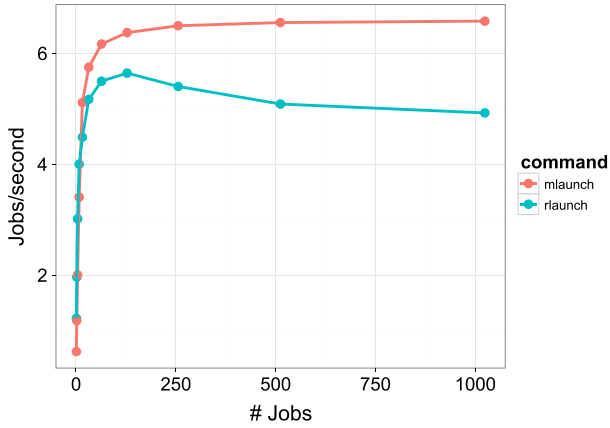

Figure 12. Throughput limits for "no-op" jobs in single stream (rlaunch) and job packing (mlaunch) modes. The results indicate that FWS v0.96 processes an upper bound of slightly higher larger than six jobs per second (~300–400 per minute), with job packing mode achieving about 20% more throughput.

range of 600 to 360,000 s (using ~50 CPUs per job), with our concurrency level set between ~50 jobs (within a single queued job) and ~2000 (for multiple queued jobs across several computing centers with high-capacity "throughput" queues).

In summary, our tests indicate that the performance of FWS should be negligible for most use cases *when each Firework runs significantly longer than a fraction of a second*. For example, the efficiency of the FWS should be over 99% for a 100-s task if the number of tasks per workflow is less than 1000 and the concurrency is kept low. If there are several Workflows in the database or multiple independent FireWorkers executing the jobs, performance should further improve, in some cases by over a factor of 10. FWS is, however, likely *not* appropriate for large-scale and concurrent "stream processing" applications of small tasks (e.g. 1-s tasks). However, we expect that most scientific use cases will not involve such extreme cases of small task sizes, and the overhead of FWS in typical use cases (>100-s tasks) will be well within acceptable limits.

5. APPLICATION TO COMPUTATIONAL MATERIALS SCIENCE AND CHEMISTRY

The FWS team has communicated with users in mathematical modeling, genome sequencing, multiscale modeling, machine learning, and computer graphics; however, it is difficult to trace actual usage of an open-source code (i.e., we do not require users to register to download the code, and we do not require connection to a centralized service to use the code). In this section, we present further details on ways in which the FWS software has been used in practice to date.

5.1. The Materials Project

The Materials Project [3] has used FWS to run over 165,000 materials workflows totaling over 1.1 million individual jobs ("Fireworks") at the NERSC. The Materials Project is a multi-institution collaboration that serves as a science gateway [24], with the computed data now serving a community of over 10,000 registered users that use the Materials Project data to guide their research [25, 26]. Over 40 million CPU-hours worth of computation have been computed with FWS on a nearly daily basis for almost 1.5 years, demonstrating its capability as a production software for a scientific center. Several types of workflows used in production by the Materials Project are available in the open source MPWorks repository (http://www.github.com/materialsproject/MPWorks).

Some of the methods through which the Materials Project uses FWS are the subject of previous [24,27,3] and upcoming [28] papers. In particular, FWS is used to distribute millions of "medium range jobs" (ranging from 1 to 1000 CPU-hours) over the PBS queuing system (Section 3.2) at two

DOI: 10.1002/cpe

separate clusters at the NERSC national supercomputing resource. The worker-specific settings (Section 3.1) are used to automatically distribute jobs with lower memory requirements to the "Hopper" system, whereas the higher memory jobs are distributed to the "Mendel" system. The Materials Project has furthermore designed a system whereby new workflows can be added via a REST interface or through the web, and those workflows (involving multiple and separate MPI jobs with dependencies) get automatically executed and analyzed for display on the website. This procedure allows what was once a specialized task involving writing complex input files and copying files and directories over the period of a week to a one-click operation that can be performed by unspecialized researchers and scaled to very large throughput.

While the Materials Project includes several types of prototypical workflows, we present as an example a workflow to calculate elastic tensors. The elastic tensor is a 6×6 tensor describing the response of a material to external forces within its elastic limit. The workflow to generate elastic tensor data consists of three stages: a set of quantum mechanical density functional theory (DFT) calculations, systematic data validations, and final data management tasks. The overall procedure is diagrammed in Figure 13 and is further described in Reference [28].

Starting with an input structure taken from the Materials Project database, the first step of the workflow is to reoptimize the structure at higher accuracy using a DFT calculation (~250 CPU-hours). The next step of the workflow is a custom FireTask that parses the output and dynamically generates 24 additional runs via the "additions" options in the FWAction (Section 3.4). In principle, the number of additional runs could be nontrivial and structure-dependent depending on the obtained symmetry of the structure reoptimization, and this would be well supported by dynamic actions in FWS that allows programmatically modifying the workflow. In the case of the Materials Project, the number of new runs is always held at 24. The 24 additional runs are executed in parallel at the NERSC supercomputing center (~300 CPU-hours per run). At the NERSC, jobs are submitted to specialized high-throughput queues that support a large number of independent parallel submissions with small parallelism (e.g., 250 concurrent jobs with 48 cores each); job packing is not used.

The next step of the workflow is to fit an elastic tensor to the results of all 24 calculations; this step depends on all jobs completing. The fitted elastic tensor then undergoes multiple consistency checks

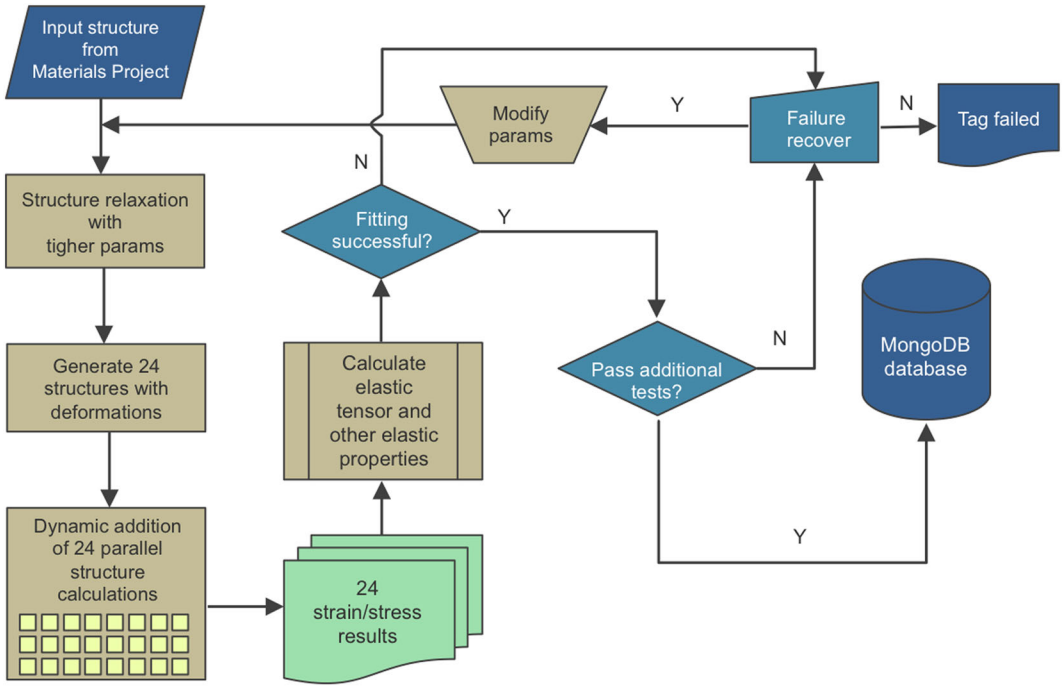

Figure 13. Workflow to compute the elastic tensor of materials as implemented for the Materials Project. Further details of this workflow are given in Reference [28].

DOI: 10.1002/cpe

[28] to identify signs of error. For structures failing any of the consistency checks, decisions are made individually on whether to rerun the calculations with a different set of parameters (e.g., with a higher *k*-points density). A similar workflow with modified parameters will be submitted automatically if a rerun decision is made. The final step in the workflow is to store the computed results in a MongoDB database. The validation and data management stages of the workflow are implemented as a series of sequential FireTasks, which are flexible to modification of validations rules and data storage options.

We have thus far calculated elastic tensors for more than 1200 materials. This represents only a small portion of materials for which more basic workflows have been run for the Materials Project. However, this database already covers almost five times greater structures than all known experimental data and is to our knowledge the largest collection of complete elastic tensors for inorganic materials.

5.2. The Electrolyte Genome

Part of the Joint Center for Energy Storage Research [29], the Electrolyte Genome is aimed at molecular discovery by computing the properties of tens of thousands of molecules. While the scientific application and physics are quite different from the Materials Project, from a workflow standpoint, these two projects are similar, and the Electrolyte Genome has to date used FWS to complete over 10 million CPU-hours worth of calculations. Here, we present one example of a molecule workflow and how modifying the workflow graph in real-time via dynamic actions is useful. Further examples of FWS usage for the Electrolyte Genome will be presented in References [30, 31].

The electrochemical window of a potential electrolyte or redox active molecule for energy storage determines its operating voltage range and thus its compatibility with various anodes and electrolytes. The calculation of electrochemical windows involves three branches of computation representing different charge states; within each branch is a sequence of dependent calculations that include a geometry optimization, a vibrational frequency, and a single point energy calculation (Figure 14). One complication in running these calculations in a high-throughput mode is the possibility of a calculation that results in imaginary frequencies, which requires further follow-up calculations with stricter parameter settings and additional computing time to fix. In the field, this procedure is typically detected and corrected manually by inspecting output files, making input file modifications, and resubmitting the job. However, such manual intervention is not possible in a high-throughput mode with tens or hundreds of thousands of molecules that involve thousands of such failures. Thus, the Electrolyte Genome uses a FireTask to detect the imaginary frequencies in the output file, and—if present—uses the "*detours*" feature of FWS (Section 3.4) to automatically set up additional runs with the appropriate modifications using a rule-based strategy. These dynamically created runs are automatically queued and executed by FWS as needed without any human intervention.

The development of such "self-healing" workflows in the Electrolyte Genome is also aided by the centralized file-tracking feature of FWS (Section 3.9). With one command, users are able to check the last few lines of the output files of all failed jobs scattered across multiple computing centers. This allows the team to detect new types of errors embedded in the output files of thousands of jobs and develop further rules for patching failed jobs, which are then embedded back in the workflows. Because the data are stored in a database, statistics of job errors can also be compiled to prioritize the most egregious failures.

5.3. Examples from individual research groups

In addition to broad collaborations such as the Materials Project and the Joint Center for Energy Storage Research, FWS is also currently used by several independent research groups. For instance, the Materials Virtual Lab (led by one of the authors) has used FWS to automate extremely long-running molecular dynamics simulations over computational resources with short walltimes. In this use case, *ab initio* molecular dynamics (AIMD) simulations that take weeks to finish have been automated at the San Diego Supercomputing Center using individual jobs that are restricted to

DOI: 10.1002/cpe

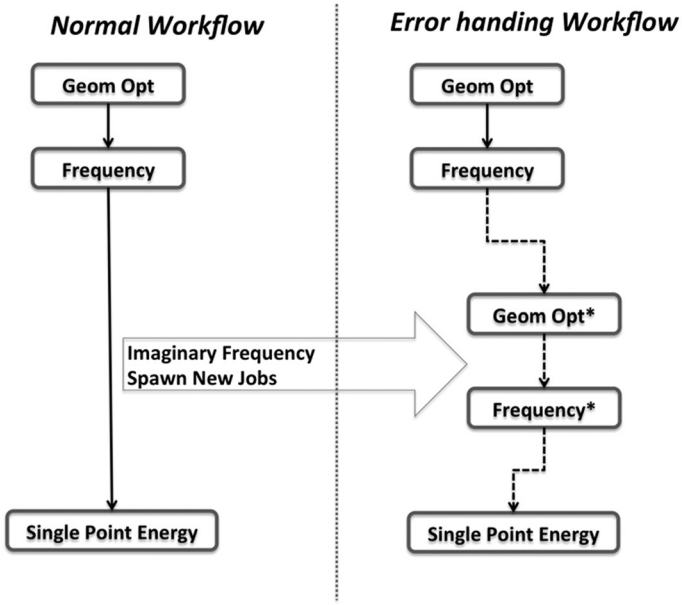

Figure 14. Illustration of a portion of a "normal workflow" (left) and automatically modified workflow graph (right) in case an imaginary frequency (failure) is detected. The FireTask of the workflow itself is programmed to automatically create more steps as needed to correct the calculation ("detour" action of FWAction). Because each job requires its own queue submission to comply with walltime queuing limits, this saves considerable manual work in setting up and queuing additional jobs for failed cases in a high-throughput setting.

walltimes of 8 h. These AIMD workflows rely on the complementary application of Custodian [32] (http://pythonhosted.org/custodian/), an error correction framework written by the authors, to execute a graceful termination and check-pointing of a run just before the walltime, followed by the dynamic addition (Section 3.4) of a continuation AIMD workflow using the results from the previous run. This workflow is depicted in Figure 15.

In essence, by merging a checkpointing routine with dynamic workflow addition, FWS has enabled the creation of "one-click" AIMD simulations, where the user only needs to initial a workflow and check the results only upon its successful completion. This is in sharp contrast to the conventional practice of running AIMD simulations, which requires human intervention to terminate and resubmit runs manually. For example, a single study might involve simulations at four temperatures, with three human restarts per day over the period of 7 days to complete one system. Assuming only 5 min of effort per restart, this totals 7 h of human intervention per materials system. FWS has thus enabled the Materials Virtual Lab to run a very large number of such AIMD simulations with minimal human resources.

As a final example, the Institute of Condensed Matter and Nanosciences group at Université Catholique de Louvain (three of the authors) have developed an "Abipy" Python framework to their popular community DFT code, ABINIT [33]. Here, FWS is used as a centralized control point that interacts with the queuing systems distributed over several European computing centers. The FWS database is used to track a large number of Abipy tasks using information within the central database. Of particular usefulness to the team is the ability to query for and address runtime failures using MongoDB. These failures can arise from code errors or failure of the targeted calculation to converge. Information about the failures is stored in the FWS database using the FWAction "stored_data" field (Section 3.4). These failure reports are sent to the user and are used to automatically perform appropriate adjustments before rerunning the workflow from the failed point. Abipy also implements the "self-healing" paradigm described in Section 5.2 and furthermore is developing and running routines in which a FireTask analyzes the best resources and parallelization strategies for its downstream job, and then dynamically creates that job with the appropriate parameters and settings that guide it to the best FireWorker. In this sense, the dynamic workflows are acting as a proxy for the Condor "matchmaking" system [18].

DOI: 10.1002/cpe

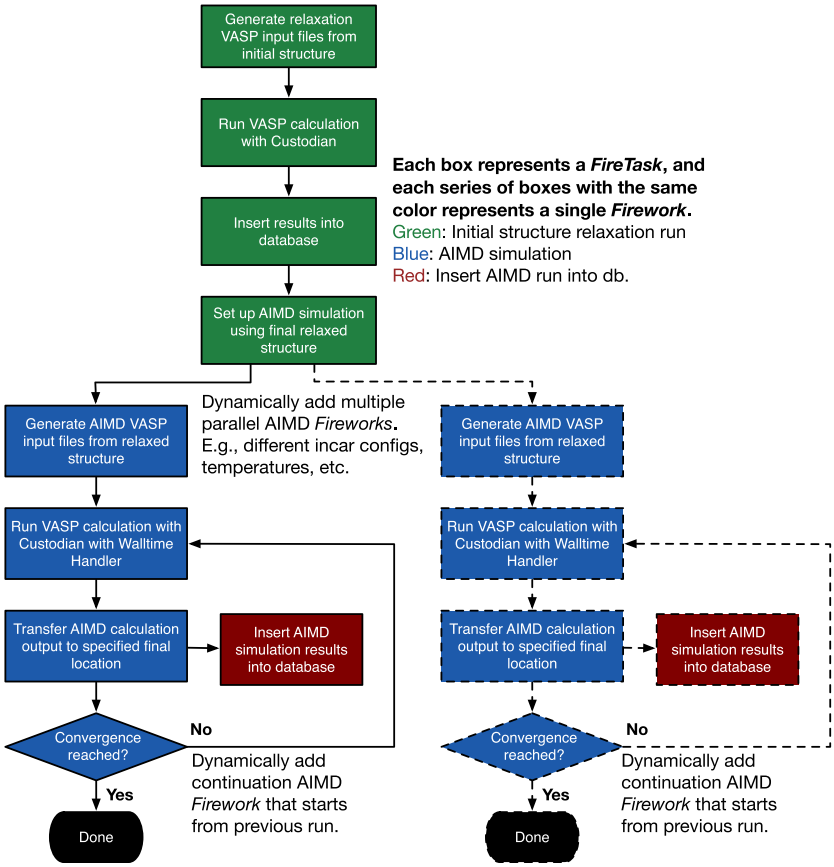

Figure 15. Schematic of AIMD workflow, showing distribution of work over several Fireworks that are separated into FireTasks. The dynamic features of FireWorks add to the workflow graph as needed at several points in the workflow.

6. RELATED WORK

Several workflow tools have been previously developed to represent and run scientific processes in a distributed environment. Many of the features presented in this work have been previously implemented over the past decade in one of more of these systems. Therefore, in this section we compare our work to these other tools and identify areas that would motivate the use of FWS.

A first comparison can be made to the Pegasus [12] and Condor DAGMan [34] systems. These are both mature systems that have been utilized for several science applications and deployed at several computing centers. Both systems contain methods to match jobs to appropriate resources and to support for several queuing systems. However, a major limitation of these frameworks is the use of *explicit* graphs, which constrain the user to workflows that can be fully defined at the time of creation. For the materials science applications discussed in Section 5, a major use case is programmatically modifying the workflow graph based on execution results. One use case is "self-healing" workflows (discussed in Section 5.2) that can automatically recover from the fairly high failure rate of DFT codes that are run with standard settings over arbitrary materials and molecules (~10%). Such functionality is to an extent present in DAGMan [34] via "recovery DAGs". However, more complex expert systems in which nuanced decisions on what to run next must be made (e.g., complex iteration and branching), even during normal operation, are typically outside the scope of rescue DAGs. For these purposes, a more dynamic workflow language is needed that can modify the workflow graph or add to it in reaction to results obtained in prior steps. The solution offered by FWS is more flexible in that one can programmatically decide how a workflow should proceed based on past results even in the absence of failures and error-handling modes.

DOI: 10.1002/cpe

More complex workflow patterns, such as hierarchical workflows and conditional iteration that are present in FWS, are supported by the Kepler [15,35], Taverna [11], and Triana [13] systems. These systems also include a graphical user interface for composing workflows, in contrast to FWS, which currently includes a graphical interface only for monitoring workflows. However, advantages in FWS include automated job bundling (Section 3.5) to improve concurrency of high-throughput jobs at supercomputing centers and duplicate subworkflow detection (Section 3.6). In particular, the duplicate detection feature of FWS allows for multiple users distributed over multiple institutions to independently submit workflows for computation. These users do not need to "check" if something has already been computed to avoid duplication; they can confidently submit their desired tasks knowing that the same workflow (i.e., same material or molecule) will be not be computed twice by FWS. Duplicated workflows steps will automatically have the output data of the first such executed job passed back without any intervention needed by the user.

We note that such "duplicate detection" can also be achieved using data-dependent workflows such as Makeflow [36]. However, a disadvantage to the data dependency implementation is its requirement on file-based outputs and correspondingly its need to have a static and well-planned organization of output files over time. This can be difficult to achieve when the outputs of a code are not flat files, or when those flat files need to be moved (e.g. archived) or reorganized in which case the dependencies are lost, or when the data to be processed are added haphazardly rather than in a planned manner. To our knowledge, the method introduced in FWS (defining tasks as JSON and using flexible JSON equivalency matching to prevent duplication) is a novel technique that provides a path forward to avoiding task duplication that is independent of the presence of data files. An example of such "file-less" duplication checking that operates via the FW spec stored in the database is presented in the FWS documentation [6].

Further comparisons of FWS can be made with Swift [19] and PyDFLOW [37], which both are systems that distribute tasks in a parallel manner. Both of the latter systems allow tasks to be programmed in an expressive way and can prevent duplicated effort because they use data dependencies. Furthermore, these systems are designed for job bundling in a manner that should achieve better performance than that of FWS (Section 4). However, these systems lack a centralized database for tracking, fixing, and rerunning failures over long time periods as well as job provenance data for completed jobs. Thus, we believe these systems are better suited to completing a well-defined working set (more in-line with many-task computing [2]) but are more poorly suited to continuously adding and monitoring workflows over the period of years, for which provenance and a centralized database are a critical practical need.

Finally, tools that are geared more towards data-analysis pipelines include Galaxy [38], KNIME [39], and IPython [7,40]/NetworkX [41] and allow beginning users to compose their analysis tasks into a logical sequence. While these tools continue to grow in capability and provide convenient graphical tools, they do not include concurrent execution features (i.e., job bundling) or failure tracking as is needed for more heavyweight jobs and as is implemented in FWS. For example, none of these systems use a "heartbeat" to detect what jobs might be lost during execution (Section 3.3).

Finally, we note that FWS may be practically convenient for many users for several reasons. For example, unlike systems such as Trident [42], FWS does not require any server-side process to run in the background at computing centers. Thus, FWS can often be installed without needing explicit support or agreement from a computing facility, as has been demonstrated at the NERSC, the San Diego Supercomputing Center, and several research group clusters. In addition, FWS passes on many of the attractive usability features of MongoDB to the workflow domain. For example, the language to modify and append to workflows is based heavily on MongoDB's syntax for updating JSON documents. Furthermore, users can leverage MongoDB's powerful query syntax (including range searches, regular expressions, and dictionary matching) to search directly on workflow documents without invoking an object-relational mapper or understanding complex joins. It is straightforward to "check the status of workflows with tags 'A' and 'B' and for which parameter 'C 'is less than 'X'", or to "rerun all failed jobs that were updated in the last 3 days". Such paradigms were, for example, used by the Materials Project to quickly identify and rerun

DOI: 10.1002/cpe

targeted sets of failed jobs. Furthermore, one can also combine this powerful query syntax with features like file tracking (Section 3.9), thereby allowing centralized monitoring of output files status for a subset of jobs. While these features are not related to fundamental issues such as job bundling performance or workflow model, they grew from challenges in implementing high-throughput materials screening at scale (over 40 million CPU-hours, over 1 million jobs for Materials Project) in a long-term project (over 1.5 years of nearly continuous computing, with addition of new workflow types, new materials, and several workflow bug-fixes added on-the-fly). We believe these features will benefit not only for the several applications that have already deployed FWS (Section 5) but also other science applications requiring long-term job management of high-throughput computations involving complex, dynamic workflows and become a useful contribution to the scientific Python community.

7. FUTURE WORK AND CONCLUSIONS

Because FWS is reasonably well tested in production, we have been able to quickly identify a number of potentially fruitful areas for future work. In practice, we have observed the major limitation in FWS to be the combination of the "pull" procedure from a central database server with strict network security on some computing centers, as described in Section 3.10. This could also be overcome by installation of appropriate network proxies, for example, using *nginx*.

Another limitation of FWS is its dependence on the Python programming language. While FWS can be used through the command line without any Python programming, it is more difficult to achieve the rich dynamic actions or control logic presented in Section 3.4 without coding a Python function. Other features, such as customized duplicate checking, require implementing Python objects. However, Python is very widely used in scientific computing and also one of the easiest (general-purpose) languages to learn.

FireWorks is explicit by design and does not interact with grid middleware tools[43], for example, automated tools for data movement or resource allocation. Indeed, one of the goals of FWS is to reduce dependence on flat files and increase usage of database-driven workflows. While FWS provides FireTasks for file transfer via secure shell (SSH), all data movement must be specified explicitly. We note that other packages already support robust integration with existing middleware tools [12,44, 45]. Thus, we expect that FWS will be more attractive to users who want to explicitly control data flow. As mentioned in Section 2.1, one method to integrate FWS with existing tools is to simply call the *rlaunch* pull script from within other workflow systems.

Future development will concentrate on making the FWS database more accessible to new users and to those who want to *monitor* workflows without necessarily taking an active part in workflow submission or management. For example, we plan to redesign the prototype frontend interface in a friendlier manner that allows one to perform basic actions such as rerunning a job and includes robust job reporting information.

In conclusion, we have described FWS, a new workflow management system with novel routines for failure detection, job packing, duplicate detection, and other tools to facilitate running high-throughput jobs at supercomputing centers. As the needs for data-driven science grow and large supercomputing centers supplant smaller computing resources for economic and energy-efficiency considerations, the need for software to manage large number of jobs over long periods at these centers will continue to grow. FWS is an open-source code already driving large numbers of results in the materials science and chemistry communities, and we hope that further improvements will increase its usefulness to the general scientific community.

8. CODE EXAMPLES

Note: all code is further explained in the FWS official documentation [6], which also contains many more examples than that presented subsequently. In addition, the MPWorks package (http://www.

DOI: 10.1002/cpe

github.com/materialsproject/MPWorks) maintained by the Materials Project as well as the fireworks-vasp package (https://github.com/materialsvirtuallab/fireworks-vasp) maintained by the Materials Virtual Lab contain production workflows for chemistry.

```
from fireworks import Firework, Workflow, LaunchPad, ScriptTask
from fireworks.core.rocket_launcher import rapidfire

# set up the LaunchPad and reset it (first time only)
launchpad = LaunchPad()
launchpad.reset('', require_password=False)

# define the individual FireWorks and Workflow
fw1 = Firework(ScriptTask.from_str('echo "To be, or not to be,"'))
fw2 = Firework(ScriptTask.from_str('echo "that is the question:"'))
wf = Workflow([fw1, fw2], {fw1:fw2}) # set of FWs and dependencies

# store workflow in LaunchPad
launchpad.add_wf(wf)
# pull all jobs and run them locally
rapidfire(launchpad)
```

Code Example 1: Python code for resetting the database, adding a two-step Workflow and executing it locally. Note that many more options exist, including setting the database configuration, adding names and tags to the FWS, alternate dependency specifications, and other launch modes.

```
lpad reset
lpad add_scripts 'echo "To be, or not to be"' 'echo "that is the question:"'
rlaunch rapidfire
```

Code Example 2 Performing the same tasks as Code Example 1, however, in this instance using shell commands typed directly into a terminal. Many options are not presented for clarity. A full description of Launchpad commands is available via “lpad –h”, and includes methods for adding, monitoring, and fixing workflows.

```
from fireworks.core.firework import FWAction, Firework, FireTaskBase
class FibonacciAdderTask(FireTaskBase):

    _fw_name = "Fibonacci Adder Task"

    def run_task(self, fw_spec):
        smaller = fw_spec['smaller']
        larger = fw_spec['larger']
        stop_point = fw_spec['stop_point']

        m_sum = smaller + larger
        if m_sum < stop_point:
            print('The next Fibonacci number is: {}'.format(m_sum))
            # create a new Fibonacci Adder to add to the workflow
            new_fw = Firework(FibonacciAdderTask(),
                              {'smaller': larger, 'larger': m_sum, 'stop_point':
stop_point})
            return FWAction(stored_data={'next_fibnum': m_sum}, additions=new_fw)

        else:
            print('With {}, have now exceeded our limit'.format(m_sum))
            return FWAction(stored_data={'final_fibnum': m_sum})
```

Code Example 3: Implementation of a FireTask that uses control flow and dynamic actions. In particular, the task performs a sum of elements in the spec, and then uses logic programmed in Python

DOI: 10.1002/cpe

either to dynamically define and add a new Firework just-in-time or to return a result. Further explanation of this code (in the context of a tutorial) is presented in the FWS documentation [6].

ACKNOWLEDGEMENTS

Work at the Lawrence Berkeley National Laboratory was supported by the Assistant Secretary for Energy Efficiency and Renewable Energy, under Contract No. DE-AC02-05CH11231. The FWS software was funded by the Materials Project (DOE Basic Energy Sciences Grant No. EDCBEE). Further funding was provided by the Joint Center for Energy Storage Research (JCESR), an Energy Innovation Hub funded by the US Department of Energy, Office of Science, Basic Energy Sciences. Additional work was supported by the Université catholique de Louvain through the "Fonds d'appui à l'internationalisation". G. H. acknowledges support by the European Union Marie Curie Career Integration (CIG) grant HTforTCOs PCIG11-GA-2012-321988. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the US Department of Energy under Contract No. DE-AC02-05CH11231. We thank M. Hargrove, D. Waroquiers, W. D. Richards, F. Brockherde, B. Foster, and W. Scullin for their contributions to the project. We thank L. Ramakrishnan and M. Haranczyk for their helpful comments on the manuscript.

REFERENCES

1. Bell G, Hey T, Szalay A. "Beyond the data deluge," *Science* (*80-*.) 2009; 1297–1298.
2. Raicu I, Foster IT, Zhao Y. "Many-task computing for grids and supercomputers," in *2008 Workshop on Many-Task Computing on Grids and Supercomputers*, 2008; 1–11.
3. Jain A, Ong SP, Hautier G, Chen W, Richards WD, Dacek S, Cholia S, Gunter D, Skinner D, Ceder G, Persson KA. "Commentary: The Materials Project: a materials genome approach to accelerating materials innovation," *APL Mater.* 2013; **1**(1):011002.
4. Livny M, Basney J. "Deploying a high throughput computing cluster," in High Performance Cluster Computing: Architectures and Systems, Volume 1. Prentice Hall, 1999.
5. "FireWorks workflow software," 2014. [Online]. Available: http://www.github.com/materialsproject/fireworks [accessed on 4 September 2014].
6. "FireWorks documentation," 2014. [Online]. Available: http://pythonhosted.org/FireWorks/.
7. Perez F, Granger BE, Hunter JD. "Python: an ecosystem for scientific computing," *Computing in Science & Engineering* 2011; **13**(2):13–21.
8. Langtangen HP. Python Scripting for Computational Science, Vol. **3** (3rd edn). Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.
9. Oliphant TE. "Python for scientific computing," *Computing in Science & Engineering* 2007; **9**(3):10–20.
10. "MongoDB." [Online]. Available: http://www.mongodb.org [accessed on 4 September 2014].
11. Oinn T, Addis M, Ferris J, Marvin D, Senger M, Greenwood M, Carver T, Glover K, Pocock MR, Wipat A, Li P. "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics* 2004; **20**(17):3045–54.
12. Deelman E, Singh G, Su M, Blythe J, Gil Y, Kesselman C, Mehta G, Vahi K, Berriman GB, Good J, Laity A, Jacob JC, Katz DS. "Pegasus: a framework for mapping complex scientific workflows onto distributed systems," *Scientific Programming* 2005; **13**:219–237.
13. Taylor I, Shields M, Wang I, Harrison A. "Visual grid workflow in Triana," *J. Grid Comput.* 2006; **3**(3–4):153–169.
14. Talia D. "Workflow systems for science: concepts and tools," *ISRN Softw. Eng.*2013; **2013**.
15. Ludäscher B, Altintas I, Berkley C, Higgins D, Jaeger E, Jones M, Lee E, Tao J, Zhao Y. "Scientific workflow management and the Kepler system," *Concurr. Comput. Pract. Exp.*2006; **18**(10):1039–1065.
16. Thain D, Tannenbaum T, Livny M. "Distributed computing in practice: the Condor experience," *Concurr. Comput. Pract. Exp.*2005; **17**(2–4):323–356.
17. Curcin V, Ghanem M. "Scientific workflow systems-can one size fit all?," in *2008 Cairo International Biomedical Engineering Conference*, 2008; 1–9.
18. Couvares P, Kosar T, Roy A, Weber J, Wenger K. "Workflow management in Condor," in Workflows for e-Science, Taylor I, Deelman E, Gannon D, Shields M (eds). Springer: London, 2007; 357–375.
19. Wilde M, Hategan M, Wozniak JM, Clifford B, Katz DS, Foster I. "Swift: a language for distributed parallel scripting," *Parallel Computing* 2011; **37**(9):633–652.
20. Cholia S, Skinner D, Boverhof J. "NEWT: A RESTful service for building High Performance Computing web applications," in *Gateway Computing Environments Workshop* (*GCE*), 2010.
21. Fielding RT. "Architectural Styles and the Design of Network-based Software Architectures," University of California: Irvine, 2000.
22. Tolosana-Calasanz R, Bañares JA, Rana OF, Álvarez P, Ezpeleta J, Hoheisel A. "Adaptive exception handling for scientific workflows," *Concurr. Comput. Pract. Exp.* 2010; **22**:617–642.
23. "Flask web framework." [Online]. Available: http://flask.pocoo.org.
24. Gunter D, Cholia S, Jain A, Kocher M, Persson K, Ramakrishnan L, Ong SP, Ceder G. "Community accessible datastore of high-throughput calculations : experiences from the Materials Project," in *MTAGS*, 2012.

DOI: 10.1002/cpe

25. Tada T, Takemoto S, Matsuishi S, Hosono H. "High-throughput *ab initio* screening for two-dimensional electride materials.," *Inorganic Chemistry* 2014; **52**(19):10347–10358.
26. Rustad J. "Density functional calculations of the enthalpies of formation of rare-earth orthophosphates," *American Mineralogist* 2012; **97**:791–799.
27. Ong SP, Cholia S, Jain A, Brafman M, Gunter D, Ceder G, Persson KA. "The Materials Application Programming Interface (API): a simple, flexible and efficient API for materials data based on REpresentational State Transfer (REST) principles," *Computational Materials Science* 2015; **97**:209–215.
28. de Jong M, Chen W, Angsten T, Jain A, Notestine R, Gamst A, Sluiter M, Ande C, Zwaag S, Curtarolo S, Toher C, Plata JJ, Ceder G, Persson KA, Asta M. "Charting the complete elastic properties of inorganic crystalline compounds," Scientific Data 2, 150009 2015.
29. Van Noorden R. "A better battery," *Nature* 2014; **507**:26.
30. Qu X, Jain A, Rajput NN, Cheng L, Zhang Y, Ong SP, Brafman M, Maginn E, Curtiss LA, Persson KA. "The electrolyte genome project: a Big data approach in battery materials discovery," *in-submission*.
31. Cheng L, Assary RS, Qu X, Jain A, Ong SP, Rajput NN, Persson KA, Curtiss LA. "Accelerating electrolyte discovery for energy storage by high throughput screening," *Journal of Physical Chemistry Letters* 2015; **6**(2):283–291.
32. Ong SP, Richards WD, Jain A, Hautier G, Kocher M, Cholia S, Gunter D, Chevrier VL, Persson KA, Ceder G. "Python materials genomics (pymatgen): a robust, open-source python library for materials analysis," *Computational Materials Science* 2013; **68**:314–319.
33. Gonze X, Amadon B, Anglade P-M, Beuken J-M, Bottin F, Boulanger P, Bruneval F, Caliste D, Caracas R, Côté M, Deutsch T, Genovese L, Ghosez P, Giantomassi M, Goedecker S, Hamann DR, Hermet P, Jollet F, Jomard G, Leroux S, Mancini M, Mazevet S, Oliveira MJT, Onida G, Pouillon Y, Rangel T, Rignanese G-M, Sangalli D, Shaltaf R, Torrent M, Verstraete MJ, Zerah G, Zwanziger JW. "ABINIT: first-principles approach to material and nanosystem properties," *Computer Physics Communications* 2009; **180**(12):2582–2615.
34. Couvares P, Kosar T, Roy A, Weber J, Wenger K. "Workflow management in condor," in Workflows for e-Science SE - 22, Taylor I, Deelman E, Gannon D, Shields M (eds). Springer: London, 2007; 357–375.
35. Altintas I, Berkley C, Jaeger E, Jones M, Ludäscher B, Mock S. "Kepler : an extensible system for design and execution of scientific workflows," in *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, 2004; 423–424.
36. Bui P, Yu L, Thain D. "Weaver: integrating distributed computing abstractions into scientific workflows using python," *Proc. 19th ACM Int. Symp. High Perform. Distrib. Comput. HPDC'10*, 2010; 636–643.
37. Armstrong TG. "Integrating Task Parallelism into the Python Programming Language," University of Chicago, 2011.
38. Goecks J, Nekrutenko A, Taylor J. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology* 2010; **11**(8):R86.
39. Berthold MR, Cebron N, Dill F, Gabriel TR, Kötter T, Meinl T, Ohl P, Sieb C, Thiel K, Wiswedel B. "KNIME: the Konstanz information miner," in *Studies in Classification*, *Data Analysis*, *and Knowledge Organization* (*GfKL 2007*), 2007.
40. Perez F, Granger B. "IPython: A System for Interactive Scientific Computing," Comput. Sci. Eng. 2007.
41. Hagberg A, Swart P, Chult D. "Exploring Network Structure, Dynamics, and Function Using NetworkX," 2008.
42. Barga R, Jackson J, Araujo N, Guo D, Gautam N, Simmhan Y. "The trident scientific workflow workbench," in 2008 *IEEE Fourth International Conference on eScience* **2008**; 317–318.
43. Foster I, Kesselman C. "Globus: a metacomputing infrastructure toolkit," *International Journal of High Performance Computing Applications* 1997; **11**(2):115–128.
44. Hoheisel A. "User tools and languages for graph-based grid workflows," *Concurr. Comput. Pract. Exp.* 2006; **18**(10):1101–1113.
45. Raman R, Livny M, Solomon M. "Matchmaking: distributed resource management for high throughput computing," in *HPDC'98 Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, 1998; **140**.
46. Shao Y, Molnar LF, Jung Y, Kussmann J, Ochsenfeld C, Brown ST, Gilbert ATB, Slipchenko LV, Levchenko SV, O'Neill DP, DiStasio RA, Lochan RC, Wang T, Beran GJO, Besley NA, Herbert JM, Lin CY, Van Voorhis T, Chien SH, Sodt A, Steele RP, Rassolov VA, Maslen PE, Korambath PP, Adamson RD, Austin B, Baker J, Byrd EFC, Dachsel H, Doerksen RJ, Dreuw A, Dunietz BD, Dutoi AD, Furlani TR, Gwaltney SR, Heyden A, Hirata S, Hsu C-P, Kedziora G, Khalliulin RZ, Klunzinger P, Lee AM, Lee MS, Liang W, Lotan I, Nair N, Peters B, Proynov EI, Pieniazek PA, Rhee YM, Ritchie J, Rosta E, Sherrill CD, Simmonett AC, Subotnik JE, Woodcock HL, Zhang W, Bell AT, Chakraborty AK, Chipman DM, Keil FJ, Warshel A, Hehre WJ, Schaefer HF, Kong J, Krylov AI, Gill PMW, Head-Gordon M. "Advances in methods and algorithms in a modern quantum chemistry program package." *Physical Chemistry Chemical Physics* 2006; **8**(27):3172–91.

DOI: 10.1002/cpe